L1VM - JIT-compiler

by Stefan Pietzonke

https://midnight-coding.de

12. April 2024

Abstract

In this paper I will show how my JIT-compiler in the L1VM works. It uses the libasmjit library for compiling the bytecode into machine code.

1 Intro

The L1VM has 256 registers for int64 and double numbers. So even the most complicated math calculations can be done by them. The bytecode is translated by the **libasmjit** library. You have to mark the beginning and the end of the code which is compiled into machine code. This is done by inserting labels in the program. You can find examples in my jit-test programs.

1.1 The opcodes

The following opcodes can be compiled by the JIT-compiler:
addi, subi, muli, divi
addd, subd, muld, divd
andi, ori, bandi, bori, bxori
eqi, neqi, gri, lsi, greqi, lseqi
eqd, neqd, grd, lsd, greqd, lseqd

jmp, jmpi

movi, movd

2 The JIT-compiler code

The JIT-compiler function **jit_compiler** does an initialization of the labels and CPU registers at start. Then it compiles the bytecode in a loop. It first checks if there is a label at current bytecode position. And then inserts the label if needed. Then the opcode is translated into the assembly code. If the opcode is not in the list then the JIT-compiler exits with an error. If an opcode was found and translated the **run_jit** variable is set to **1** to mark it as compiled.

2.1 The saving of the assembly code

At the end of the JIT-compiler the code is saved if the **run_jit** variable is set to **1**. Here is the code part: https://github.com/koder77/l1vm/blob/master/libjit/jit.cpp

```
if (run_jit)
{
        a.ret ();               // return to main program code

        // printf ("JIT_code_ind: %lli\n", JIT_code_ind);

        if (JIT_code_ind < MAXJITCODE - 1) // JIT_code_ind overflow fix!!
        {
            // create JIT code function
```

```
               JIT_code_ind++;

               Func funcptr;

               // store JIT code:
               Error err = rt.add (&funcptr, &jcode);
               if (err == 1)
               {
                   printf ("JIT␣compiler:␣code␣generation␣failed!\n");
                   return (1);
               }

               JIT_code[JIT_code_ind].fn = (Func) funcptr;
               JIT_code[JIT_code_ind].used = 1;
               #if DEBUG
                   printf ("JIT␣compiler:␣function␣saved.\n");
               #endif

               return (0);
           }
           else
           {
               printf ("JIT␣compiler:␣error␣jit␣code␣list␣full!\n");
               return (1);
           }
     }
     return (0);
```

2.2 The run of the code

The compiled code is run by the **run_jit** function:

```
   extern "C" int run_jit (S8 code ALIGN, struct JIT_code *JIT_code)
   {
           #if      DEBUG
                   printf ("run_jit:␣code:␣%lli\n", code);
           #endif

           if (code < 0 || code >= MAXJITCODE)
           {
                   printf
("JIT␣compiler:␣FATAL␣ERROR!␣code␣index␣%lli␣out␣of␣range!!!\n", code);
                   return (1);
           }

           if (JIT_code[code].used == 0)
           {
                   printf
("JIT␣compiler:␣FATAL␣ERROR!␣code␣index␣%lli␣not␣compiled!\n", code);
                   return (1);
           }

           Func func = JIT_code[code].fn;

       #if DEBUG
           printf ("run_jit:␣code␣address:␣%lli\n", (S8) func);
```

```
    #endif

    if (func == NULL)
    {
            printf ("JIT␣compiler:␣FATAL␣ERROR!␣NULL␣pointer␣code!!!\n");
            return (1);
    }

    // call JIT code function, stored in JIT_code[]
    JIT_code[code].fn();
    return (0);
}
```

2.3 The cleanup

The generated code is freed by the **free_jit_code** function at program end:

```
extern "C" int free_jit_code (struct JIT_code *JIT_code, S8 JIT_code_ind)
{
        /* free all JIT code functions from memory */

        S4 i;

        if (JIT_code_ind > -1)
        {
                for (i = 0; i <= JIT_code_ind; i++)
                {
                        rt.release((Func *) JIT_code[i].fn);
                }
        }

        return (0);
}
```

2.4 Summary

As it can be seen the JIT-compiler is not difficult to understand. If you know how the needed assembly is used. I did use the **64 bit** assembly opcodes in the JIT-compiler. So there are no **32 bit** opcodes used. The most difficult part was the binding between the VM bytecode and the JIT-compiler opcodes assembly code. I also did contact the author of **libasmjit**, he could help me by my code. If you have any questions you can write me: **spietzonke@gmail.com**. I hope this short paper was useful.

3 The usage in Brackets code

Here I will show how to use the JIT-Compiler in Brackets code. There are two JIT-compilers: one for "x86_64" and one for "AARCH64" (ARM 64 bit). The supported opcodes for "x86_64" are:

addi, subi, muli, divi
addd, subd, muld, divd
andi, ori, bandi, bori, bxori
eqi, neqi, gri, lsi, greqi, lseqi
eqd, neqd, grd, lsd, greqd, lseqd

jmp, jmpi
movi, movd

Only this opcodes are allowed in the code to compile it by the JIT-compiler.

3.1 Mark the code with labels

The JIT-compiler needs to know which part of the code should be compiled. You have to set a start and an end label. Here is the "jit-test.l1com" program: https://github.com/koder77/l1vm/blob/master/prog/jit-test.l1com

```
(main func)
    (set int64 1 zero 0)
    (set int64 1 x 23)
    (set int64 1 y 42)
    (set int64 1 z 7)
    (set int64 1 i 1)
    (set int64 1 max 100Q)
    // (set int64 1 max 800000000Q)
    (set int64 1 sum 0)
    (set int64 1 logic)
    (set int64 1 one 1)
    (ASM)
    loada zero, 0, I0
    loada x, 0, I1
    loada y, 0, I2
    loada z, 0, I3
    loada i, 0, I4
    loada max, 0, I5
    loada one, 0, I6
    loada sum, 0, I10
    loadl :jit, I40
    loadl :jit_end, I41
    // run jit compiler
    intr0 253, I40, I41, 0
:loop
    // call jit code
    intr0 254, I0, 0, 0
    // jump to following non-jit code
    jmp :next
:jit
    muli I4, I3, I20
    muli I4, I1, I21
    addi I4, I2, I22
    addi I10, I20, I10
    addi I10, I20, I10
    addi I10, I20, I10
    addi I10, I20, I10
    addi I10, I20, I10
    addi I10, I21, I10
    addi I10, I21, I10
    addi I10, I21, I10
    addi I10, I21, I10
    addi I10, I21, I10
    addi I10, I22, I10
    addi I10, I22, I10
    addi I10, I22, I10
    addi I10, I22, I10
```

```
    addi I10, I22, I10
    bandi I1, I2, I52
    bori I1, I2, I53
:jit_end
    bxori I1, I2, I54
    // store
:next
    // intr0 4, I10, 0, 0
    // intr0 7, 0, 0, 0
    load sum, 0, I30
    pullqw I10, I30, 0
    addi I4, I6, I4
    lseqi I4, I5, I30
    jmpi I30, :loop
    intr0 4, I10, 0, 0
    intr0 7, 0, 0, 0
    intr0 4, I52, 0, 0
    intr0 7, 0, 0, 0
    intr0 4, I53, 0, 0
    intr0 7, 0, 0, 0
    intr0 4, I54, 0, 0
    intr0 7, 0, 0, 0
    intr0 255, 0, 0, 0
    (ASM_END)
(funcend)
```

3.2 Run the JIT-compiler

This block saves the start and end labels. And runs the JIT-compiler:

```
loadl :jit, I40
loadl :jit_end, I41
// run jit compiler
intr0 253, I40, I41, 0
```

After this the start of the loop is set:

```
:loop
    // call jit code
    intr0 254, I0, 0, 0
    // jump to following non-jit code
    jmp :next
:jit
```

This line calls the JIT-code at index "I0" (the first JIT-compiled block): " intr0 254, I0, 0, 0". After this the program skips the normal bytecode with the "jmp :next" statement. This is needed because the code is already executed as compiled code! There can be multiple blocks with marked code. You then have to set a new index "1" at the end of the JIT-compiler call: "intr0 253, I40, I41, 1". You can also use the interrupt names defined in the include file "intr.l1h":

```
>> JIT-compiler
#func run_jit_comp (SLAB, ELAB, CODE) :(253 SLAB ELAB CODE intr0)
#func run_jit_code (CODE) :(254 CODE 0 0 intr0)
```

If you use an opcode in your code which is not in the JIT-compiler, then you will get a runtime error! The JIT-compiler generates very fast code and can run the code blocks near the speed of a C program with the same algorithm. Here is a benchmark: https://midnight-coding.de/blog/software/l1vm/2023/05/07/L1VM-benchm-loop.html